A Virtual Multi-Channel GPU Fair Scheduling Method for Virtual Machines

Huailiang Tan, Yanjie Tan, Xiaofei He, Kenli Li, and Keqin Li, *Fellow, IEEE*

Abstract—In modern virtual computing environment, the 2D/3D rendering performance and parallel computing potential of GPU (graphics processing unit) must be fully exploited for multiple virtual machines (VMs). Existing GPU virtualization techniques are unable to take full advantage of a GPU's powerful 2D/3D hardware-accelerated graphics rendering performance or parallel computing potential, or it has not been considered that the internal resources of a GPU domain are fairly allocated between VMs with different performance requirements. Therefore, we propose a multi-channel GPU virtualization architecture (VMCG), model the corresponding credit allocating and transferring mechanisms, and redesign the virtual multi-channel GPU fair-scheduling algorithm. VMCG provides a separate V-Channel for each guest VM (DomU) that competes with other VMs for the same physical GPU resources, and each DomU submits command request blocks to its respective V-Channel according to the corresponding DomU ID. Through the virtual multi-channel GPU fair-scheduling algorithm, not only do multiple DomUs make full use of native GPU hardware acceleration, but the fairness of GPU resource allocation is significantly improved during GPU-intensive workloads from multiple DomUs running on the same host. Experimental results show that, for 2D/3D graphics applications, performance is close to 96 percent of that of the native GPU, performance is improved by approximately 500 percent for parallel computing applications, and GPU resource-allocation fairness is improved by approximately 60-80 percent.

Index Terms—Credit modeling, fair scheduling, GPU virtualization, hybrid application workloads, virtual multi-channel

1 Introduction

In general, GPU is used to accelerate graphics displays. With GPU virtualization techniques, not only can DomUs[1] on personal computers (PCs) or servers share a physical GPU adapter that has multiple display ports, but each DomU can also independently output graphics by binding each display port, such as in multi-screen independent display technology [6]. However, each DomU's 2D/3D graphical performance is low due to bypassing the hardware-accelerated resources of a physical GPU. To meet the requirements of virtual office environments and virtualization application scenarios that need high graphic processing power [1], high-performance GPU virtualization is urgent [2]. On the other hand, a GPU plays the same role as a CPU in high-performance parallel computing [16]. For promoting computing performance, some parallel tasks from multiple DomUs are deployed to GPU subsystems, leading to competition for GPU resources. It is also common that hybrid GPU-intensive application workloads (e.g., 2D/3D graphic rendering, parallel computing, and GPU hardware encoding/decoding) from multiple DomUs exclusively share the same physical GPU. Therefore, in the case of hardware-accelerating conditions, fair and stable GPU resource allocation between multiple DomUs has significant implications for VMs.

Optimally managing and fairly allocating GPU resources among numerous DomUs is an effective technique to improve the performance of GPU-intensive applications. Research efforts have been directed toward exploiting physical GPU performance in virtualized environments. We can generally divide such efforts into the following two categories: (1) enhancing graphic performance under virtualization, and (2) facilitating parallel computing under VMs. For the first category, 3D rendering is achieved by using a modular driver framework in guest DomUs [5]. Since the native GPU is not fully exploited, this modular method brings low 2D/3D rendering performance. The literature [7] compares GPU virtualization with emulation and native GPU, but fair GPU resource allocation is not considered. Furthermore, GViM [8] is designed for virtualizing and managing the resources of a general-purpose system accelerated by graphics processors, but fails to consider parallel computing and hybrid application workloads. To avoid emulating rendering in the guest OS, DomUs share only one RequestQueue for graphics processing by exploiting GPU hardware rendering [3]; however, they share the same RequestQueue into which DomUs issue their GPU command request blocks, thus causing interference between DomUs. An "Agent" owned by each DomU and a scheduling controller shared by all VMs are designed in

1. A DomU is an unprivileged domain which is the counterpart to Dom0, and Dom0 is an abbrevation of "Domain 0" in Xen which is the premier domain initiated by the hypervisor on boot

- *H. Tan, Y. Tan, X. He, and K. Li are with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China. E-mail: {tanhuailiang, tanyanjiecws, S141000891, lkl}@hnu.edu.cn.*
- *K. Li is with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China and also with the Department of Computer Science, State University of New York, New Paltz, NY 12561. E-mail: lik@newpaltz.edu.*

Manuscript received 9 Dec. 2017; revised 18 July 2018; accepted 9 Aug. 2018. Date of publication 13 Aug. 2018; date of current version 16 Jan. 2019. (Corresponding authors: Huailiang Tan and Keqin Li.) Recommended for acceptance by P. Balaji. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2018.2865341

VGRIS [6]. The scheduling controller manages the agent to access GPU for achieving 3D game rendering, which only focuses on the GPU scheduling sequence when multiple DomUs concurrently generate respective workloads to the same physical GPU. Unfortunately, the fairness of GPU resource allocation is ignored, and hybrid GPU workloads between DomUs have not been considered. Moreover, the rCUDA [25], [28], [29] is organized as a client-server distributed architecture in which the sockets API is used for communication between clients and servers, and it is the first solution that is VMM-independent and production-ready for CUDA virtualization [30]. But for TCP/IP with a low-bandwidth interconnect, GPU access rate is reduced when clients send GPU command request blocks to the server to access physical GPU [31]. Then, the literature [32] proposes a framework which uses InfiniBand(IB) FDR to afford higher effective bandwidth and significantly reduce the overhead. It is the sole ready-to-manufacture framework aiming at HPC cluster environments in comparison with other virtualization solutions. But it also neglects the GPU resource allocation's fairness. GPUvm [4] provides a GPU shadow channel for each DomU. Each DomU owns two types of indexes (physical indexes and virtual indexes). These indexes occupy too much address space and make the system more complex. The BAND scheduling algorithm is designed to extend the CREDIT scheduling in GPUvm. However, a virtual machine with low priority might never access GPU when GPUvm uses the prioritization policy between DomUs, and it only inserts waiting time in the GPU context to meet the fairness required by similar workloads. In addition, the BAND scheduling does not consider the number of consumption credits for DomUs too. Thus, it is necessary to remodel CREDIT. For the second category, Shi [14] and Vu [11] only focus on achieving high-performance computing in VMs, while graphics processing and hybrid workloads are excluded, and the fairness of GPU resource allocation is not considered.

The gVirt makes breakthrough progress in full GPU virtualization [2] by proposing an architecture that concurrently shares GPU using a mediator pass-through. In gVirt, a virtual full-fledged GPU is presented to each VM which can directly access performance-critical resources with no hypervisor's intervention in many situations. So it minimizes the cost of privileged operations from guest through trap-and-emulate. The performance of GPU for DomUs can achieve up to 95 percent native performance. However, since VMs are required to serve different workloads from multiple DomUs simultaneously, GPU command request blocks of all workloads are aggregated to a uniform queue, i.e., the command request block queue in Dom0 is shared by all DomUs' virtual GPU. In this case, a request block is scheduled in its arrival sequence. GPU performance experienced by some DomUs suffers from the interference of the command request blocks of other DomUs. It is obvious that a shared FCFS (first come first serve) queue in Dom0 cannot meet the requirement of fairness. For example, we deployed three same GPU-intensive applications (PassMark 3D [22]) to three DomUs, and set the same weight (i.e., 1:1:1) for each DomU in gVirt. They obtain 81-122 FPS (frames per second), 109-284 FPS, and 30-46 FPS respectively, which is clearly unfair and unstable in GPU resource allocation among DomUs. In summary, existing GPU virtualization solutions cannot restrict the amount of GPU resources consumed by individual DomUs (each DomU may issue as many GPU command request blocks as it wants), and also does not consider the mutual interference between DomUs and the fairness of GPU resource allocation.

In this paper, we propose a virtual multi-channel GPU virtualization architecture (VMCG) that not only provides a fully functional virtual GPU for each DomU but also fairly assigns GPU resources to each DomU according to the requirements of GPU application workloads. Moreover, a V-Channel, directly using a DomU ID to eliminate index transferring overhead and save memory space, is provided for each DomU. We model the corresponding credit allocating and transferring mechanisms, and design a virtual multi-channel GPU fair-scheduling algorithm. Furthermore, VMCG is completely based on a software-level design, and does not require any additional special hardware support. Finally, we verify the effectiveness of VMCG through a series of experiments, including 2D/3D graphics processing applications, parallel computing, and even hybrid GPU-intensive workloads.

The main contributions of our work are as follows.

We propose VMCG, a virtual multi-channel GPU virtualization architecture that provides a fully functional virtual GPU for each DomU. The architecture also allows GPU resources to be fairly assigned to each V-Channel. Additionally, VMCG separates the unified management of all running DomUs in Dom0, and builds an independent V-Channel mapping in the GPU for each DomU.

We model the credit allocation and transferring mechanisms between virtual GPUs (vGPUs), and design a vGPU fair-scheduling algorithm that allocates GPU resources according to the actual demands of each DomU.

VMCG is implemented by extending and revising gVirt in Xen. A series of experiments, including 2D/3D graphics-processing applications, parallel computing, and even hybrid GPU-intensive workloads, are evaluated. Experimental results demonstrate that VMCG clearly improves the fairness and stability of GPU resource allocation.

The rest of this paper is organized as follows. Section 2 describes the background and motivation. Section 3 proposes the VMCG architecture based on gVirt. Section 4 illustrates the credit model. The VMCG fair-scheduling algorithm is described in detail in Section 5. Evaluations are provided in Section 6, and related works are discussed in Section 7. Finally, Section 8 concludes the paper and Section 9 is the acknowledgments.

2 Background and Motivation

2.1 Background

At present, there are three types of GPU virtualization technologies in Xen hypervisor [12]: Device Emulation, API Forwarding, and Direct Pass-Through. To provide graphics output for virtual machines, Device Emulation consumes much CPU and memory resource because the CPU emulates a GPU's functionalities, leading to lower host performance and slow graphic rendering speed. API Forwarding intercepts API calls in the application layer and achieves corresponding functions with forwarding or simulation. But this scheme is difficult to be promoted, because the partial functions (such as virtual machines) are not considered

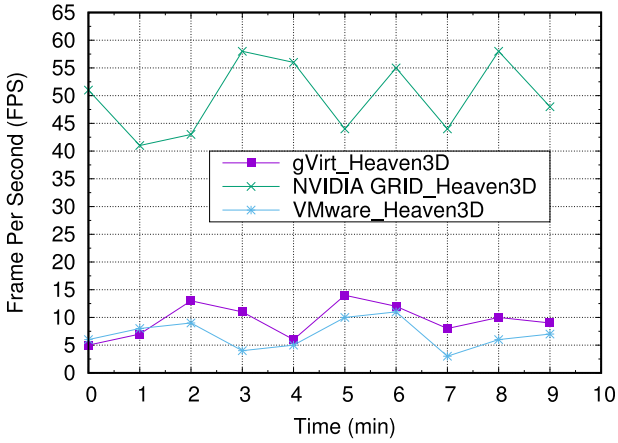


Fig. 1. The performance of heaven 3D.

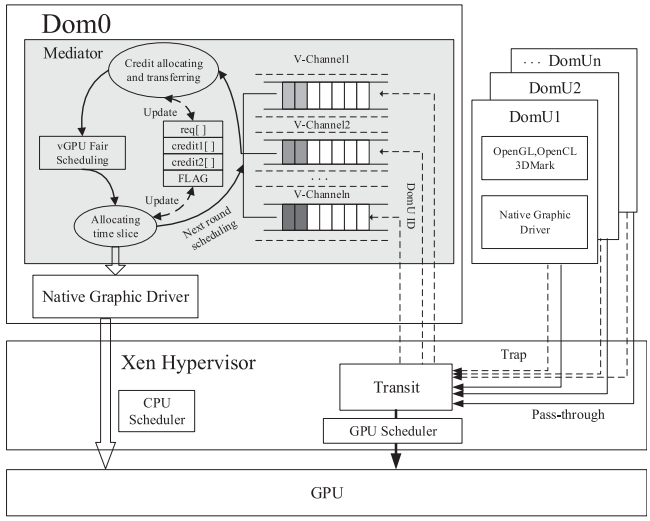


Fig. 2. VMCG architecture.

at the beginning of the design and their internal structure is also unknown. Direct Pass-Through manages the system by bypassing the virtual machine. However, it has a poor compatibility and the hardware cost is too high.

A full GPU virtualization based on Xen hypervisor, namely gVirt, has been designed [2]. gVirt also chooses the Dom0 kernel as the privileged domain to manage multiple unprivileged VMs (i.e., DomUs). In the meanwhile, unlike classical Xen supporting simply the whole I/O resource's pass-through or trap, the memory virtualization module (vMMU) of Xen hypervisor is extended to achieve the policies of trap and pass-through in gVirt, which contains EPT (extended page tables) for DomUs and PVMMU (para-virtualized memory management unit) for Dom0. The performance-critical resources of the GPU can be accessed directly by each DomU running a native graphics driver. To protect privileged resources, GPU command request blocks from graphics driver in DomUs and Dom0 are trapped to the mediator driver in Dom0 for emulation. For accessing the physical GPU and sharing it among DomUs, a hypercall and an independent GPU scheduler, which is separated from the exsiting CPU scheduler but runs concurrently, are implemented in the mediator. Thus all of the submitted DomU commands can be executed directly by the physical GPU so that the simulation of the most complex component in the GPU (i.e., the rendering engine) is avoided. The performance of a GPU within DomUs can achieve up to 95 percent of native performance. Therefore, we design VMCG based on gVirt to implement the fair allocating and scheduling of a GPU.

2.2 Motivation

Neither gVirt nor Native Xen can achieve fair GPU resource allocation with multiple DomUs. Furthermore, some other virtualization solutions, such as VMware and NVIDIA GRID, also show unstable states among DomUs, according to our analysis. Fig. 1 represents part of experimental results by running the Heaven 3D workload with these virtualization technologies in three DomUs. From the figure, it can be seen that the performance of all the three solutions fluctuate over time (more details will be disscussed in Section 6). So there is a necessity to implement a virtualization architecture that guarantees the GPU resources to be assigned fairly. According to our observation, there are three key factors that have implications on credit allocation. First, on Intel platform, system memory is used as graphic memory [2], so we analyze the graphic memory utilization with the increasing number of DomUs and data size in Rodinia [24]. Since the total graphic memory is limited, then the greater the number of DomUs, the bigger the amount of data size, the less graphic memory each DomU could be assigned. From these experiments, we conclude that graphic memory is an important factor restricting the performance of GPU computing, therefore the number of credits distributed to $DomU_i$ is related to the size of graphic memory consumed by $DomU_i$. Second, different PCI-E bandwidth occupancy between VMs also affects the number of credits because it controls the rate of data transmission between CPU and GPU domains. Finally, if a GPU command request block in a V-Channel is handed out, then a credit will be consumed. Based on the above observations, we extend the credit scheduling in VMCG so that it can achieve fair GPU resource allocation as proposed in this paper and the fairness of GPU resource allocation will be affected by the mechanism of credit distribution.

3 VMCG Architecture

Fig. 2 shows the overall architecture of VMCG, which is based on gVirt [2] to achieve the native GPU performance in VMs. Dom0 is still treated as the privileged VM and other DomUs are used as guest VMs. The mediator module of gVirt is extended in VMCG to reduce mutual interference between DomUs and improve the fairness of GPU resource allocation. It maintains an independent V-Channel that can be bound to a display port for each DomU in Dom0. The command request blocks from transit which belong to DomUs are forwarded to respective V-Channels according to the corresponding DomU ID. Furthermore, because a GPU domain is essentially a parallel computing system that owns inherent stream processors, video RAM, and an I/O bus similar to a CPU domain, for achieving fair GPU resource allocation, a similar vGPU (vGPU structure) with a virtual central processing unit (vCPU structure) is implemented in the mediator module. Thus, the command request blocks in V-Channels are distributed to the native graphics driver in Dom0 through vGPU scheduling.

TABLE 1
List of Parameters

Parameter Name	Description
$TIME_i$	the amount of time that $DomU_i$ is entitled for execution
NUM_i	the number of command request blocks from $DomU_i$'s pending queue
BW_i	the PCI-E bandwidth occupancy of $DomU_i$
$MSIZE_i$	the graphic memory size of $DomU_i$

Inspired by the idea of credit-based scheduling that is used to guarantee CPU resource scheduling fairness in Xen hypervisor, GPU resources are abstracted as credits in the VMCG. GPU resource allocation means that a certain number of credits are allocated to each DomU. The vGPU scheduler executes context switching between vGPUs, and the time slice to switch is given by the vGPU fair-scheduling algorithm, which can be measured through the credit allocating and transferring mechanisms. The number of credits for each DomU is closely related to the GPU resources needed by the DomU.

4 Credit Model

4.1 Mapping from Resources of GPU Domain to Credit

VMCG maintains an independent V-Channel for each DomU in Dom0. GPU command request blocks of DomUs are first inserted into the corresponding queue according to DomU ID; then, they are distributed to the native graphics driver in Dom0 through vGPU scheduling. Credit is the abstraction of GPU resources. VMCG fairly allocates credits for each DomU according to the command request block quantity, graphic memory size, and PCI-E bandwidth. Table 1 displays the parameters we denote. BW is the total PCI-E bandwidth, and $MSIZE$ is the total graphic memory size. PCI-E traffic can be monitored by an experimental tool, such as Intel Performance Counter Monitor (PCM), which is called pcm-pcie. According to Pharr et al. [13], Kun et al. [2], and Zheng et al. [6], the GPU resource $TIME_i$ of $DomU_i$ shows an exponential function distribution, i.e.,

$$TIME_i = \frac{e^{BW_i \times MSIZE_i} + NUM_i}{e^{BW \times MSIZE} + \sum_{j=1}^{N} NUM_j} \times TIME, \quad (1)$$

where N is the number of DomUs that generate GPU command request blocks and $TIME$ is the GPU time requirements of all DomUs for a period. Thus, the credit $CREDIT_i$, which represents abstract GPU resource requirements for $DomU_i$, can be obtained by Eq. (2)

$$CREDIT_i = \frac{e^{BW_i \times MSIZE_i} + NUM_i}{e^{BW \times MSIZE} + \sum_{j=1}^{N} NUM_j} \times CREDIT, \quad (2)$$

where $CREDIT$ is the total number of credits.

4.2 Credit Allocation

Fig. 3 shows the procedure of the credit allocation. Credit allocation causes GPU resources to be fairly exploited by all DomUs, effectively alleviating GPU resource competition between multiple DomUs. Actually, in an interval $TIME_G$ (period between two consecutive replenishment event which aims to trigger the credit reallocation strategy) of credit allocation, we find that the PCI-E bandwidth occupancy and graphic memory size of $DomU_i$ are essentially unchanged. The real consumption of credit is related to the change of GPU command request blocks in the pending queue. Therefore, we treat BW_i and $MSIZE_i$ as a constant, and the credits consumed are only related to distributing command request blocks.

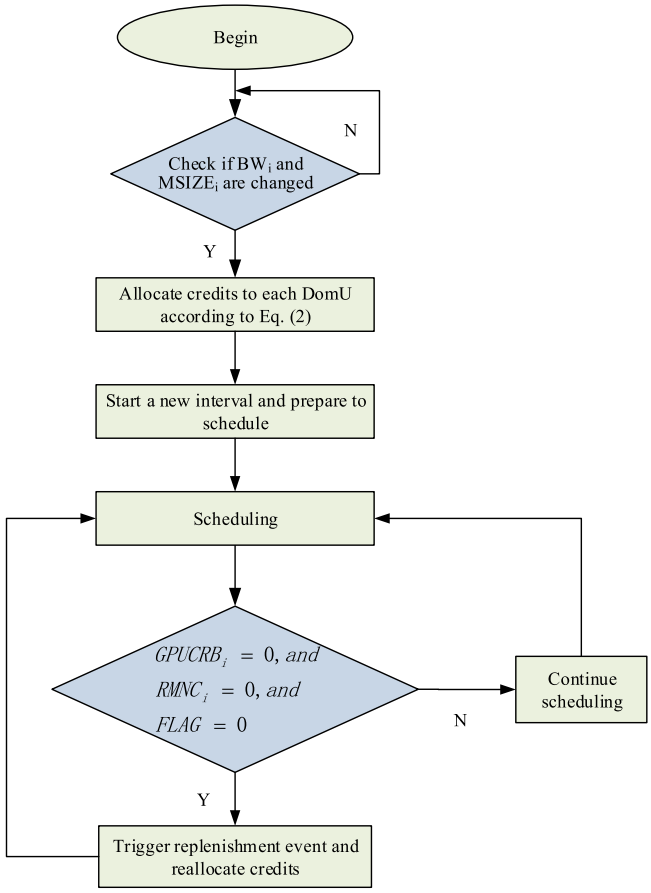


Fig. 3. Credit allocation process.

We quantify the number of credits transferring by monitoring the whole credit allocating process, which is responsible for providing all information needed by the VMCG. There are four metrics collected from the channels of all DomUs: $FLAG$ (the sparseness status of current GPU usage), $GPUCRB_i$ (the number of GPU command request blocks of $DomU_i$), $RMNC_i$ (the number of remaining credits of $DomU_i$), and $RALC_i$ (the number of reallocated credits of $DomU_i$). These metrics are used as indicators by the scheduler when dispatching GPU command request blocks, and can be calculated according to the following methods.

1) $FLAG$: This value is utilized to indicate GPU sparseness status; '0' means that the GPU is in an idle state, and '1' indicates that the GPU is in a busy state.
2) $GPUCRB_i$: We use an array, *req[n]*, to record $GPUCRB_i$. The number of GPU command request blocks of $DomU_i$ is written into array *req[n]* correspondingly. During scheduling, when a GPU command request block of $DomU_i$ enters the pending

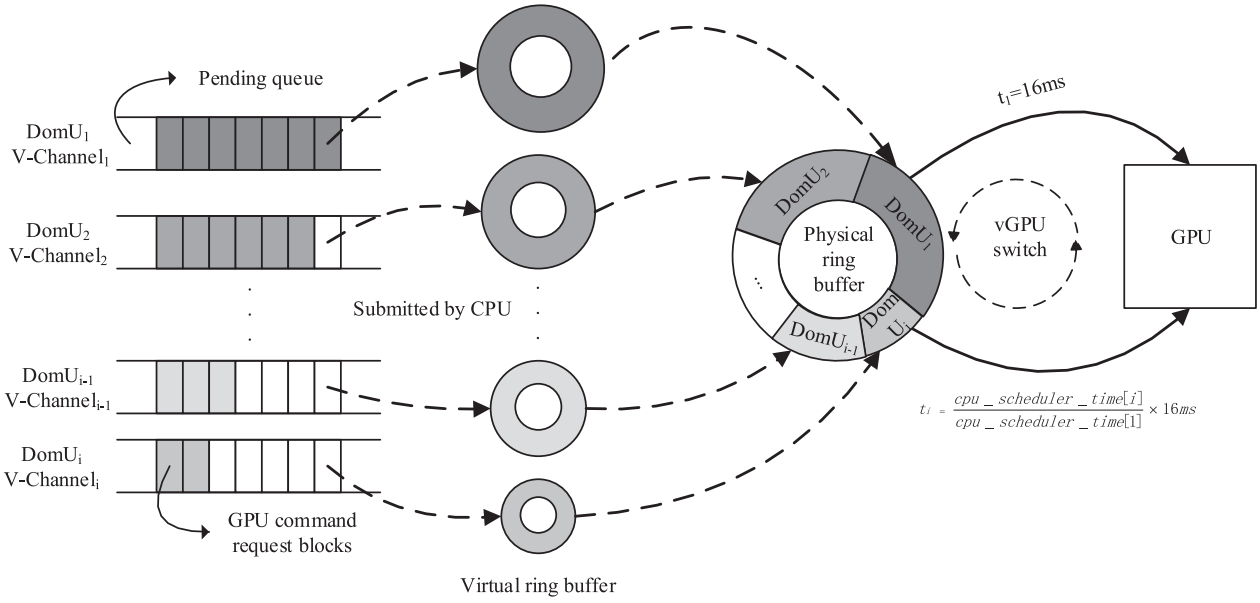


Fig. 4. Allocating time slice for DomUs.

queue in Dom0, the value of *req[i]* is increased by 1. When a request block of $DomU_i$ is dispatched from the pending queue, the value of *req[i]* is decreased by 1. Thus, we have $GPUCRB_i = req[i]$.

3) $RMNC_i$ is stored in another array, *credit1[n]. credit1 [i]* is the number of remaining credits of $DomU_i$. When the current interval ends, *credit1[i]* is decreased; if *credit1[i]* = 0, it is updated by *credit2[i]*.
4) $RALC_i$: We also use an array, *credit2[n]*, to record $RALC_i$. The value of *credit2[i]* is the number of reallocated credits for $DomU_i$ when the value of *credit1[i]* is 0. Thus, *credit2[i]* = $credit1[i] = CREDIT_i$.

GPU command request blocks of $DomU_i$ are dispatched out when $FLAG$ = 0, and the number of remaining credits for $DomU_i$ will be reduced, i.e., the value of *credit1[i]* is decreased. However, before reducing the value of *credit1[i]*, it is necessary to check whether the value of *credit1[i]* equals 0; if yes, *credit1[i]* and *credit2[i]* must be updated.

Credit allocation also contains the credit reallocation strategy, which aims to increase the utilization of GPU resources through credit reallocation. In the VMCG, the replenishment event is triggered in the following two circumstances. First, an interval has elapsed. The period $TIME_G$ is dynamically adjusted, $TIME_G$ is updated when the GPU command request blocks in the queue have been changed. We can calculate the time slice for each DomU, and then treat the time slice as a value of the $TIME_G$. Second, all credits of a DomU's pending queue are dispatched out, and there are new command request blocks incoming, and spare GPU resources exist.

5 VMCG Fair Scheduling

The VMCG uses a vGPU structure to bind the mapping from DomU to physical GPU resources. The most important resource bound in the vGPU is the ring buffer. Fig. 4 illustrates the process of fairly allocating time slice for each DomU. As shown in the middle of Fig. 4, the VMCG provides a separated ring buffer for each DomU, namely the virtual ring buffer, which is allocated in the local memory of the DomU. Each DomU appears to monopolize the ring buffer; but in fact, the GPU's physical ring buffer is shared by all DomUs. For example, the command request blocks of DomU1 in the virtual ring buffer are initially packaged into the physical ring buffer when the GPU engine is occupied by DomU1. When DomU1's command request blocks are consumed in the ring buffer, and the physical ring buffer is in the idle state, the GPU engine switches to DomU2, i.e., the next DomU owns the physical ring buffer because the command ring buffer has a producer-consumer work mode. The vGPU fair-scheduling primarily involves the following two aspects.

5.1 Fairly Allocating Scheduling Time Slices for Each DomU

In the process of vGPU scheduling, each DomU sends its request blocks from the corresponding pending queue into its virtual ring buffer. Then, the GPU fetches request blocks from the virtual ring buffer, as shown in the left side of Fig. 4. To approach the performance of a native GPU, VMCG directly uses the physical GPU engine to handle most command request blocks from DomUs. We must consider when the GPU engine switches between DomUs. Due to different GPU demands for different priority of DomUs, the weight of GPU resources is not the same among DomUs. Thus, to better serve the fair-scheduling of GPU, we initially measure the GPU occupancy weights of different DomUs. GPU commands are derived from the command ring buffer, and the more commands a DomU application sends, the longer the time slice the DomU will receive. Therefore, the DomU has a higher weight. Moreover, GPU commands are submitted through the host CPU domain. Thus, for a certain DomU, its CPU processing time and GPU processing time are proportional. Considering this relationship, we allocate the corresponding time slice for DomUs by exploiting the time of the CPU domain for addressing the command ring buffer. Actually, we can acquire the CPU scheduling time through CPU context switching, and VMCG updates the values when the switch occurs.

5.2 Virtual Multi-Channel Fair Scheduling Algorithm

Algorithm 1 shows the procedure of vGPU scheduling time slices for each DomU allocation which is illustrated from Step 1 to 13, and the rest steps represent the process of the GPU command request blocks dispatch. The array *cpu_scheduler_time[n]* is used to record the CPU scheduling time of each DomU, and VMCG updates the value of the array when CPU context switching occurs, and then sorts the array in descending order (Step 3 and 4). In addition, the value of array *DomID_Num[n]*, which saves the corresponding DomU ID, is also exchanged (Step 5). The specific steps are as follows. First, we allocate 16 ms to the maximum value of DomU in array *cpu_scheduler_time[n]*; since 16 ms is less than the transformation time perceptible for humans for graphics and images [2], we choose this value as the largest vGPU time slice. That is, the first DomU is allocated 16 ms after sorting the *cpu_scheduler_time[n]* (Step 9), and other DomUs' time slices are $t_i = (cpu\_scheduler\_time[i])/(cpu\_scheduler\_time[1]) \times 16$ ms, where t_i is the time slice of $DomU_i$ (Step 10 to 13). Second, the GPU engine is switched to another DomU whose time slice is the second largest value in array *cpu_scheduler_time[n]*. Finally, the values in the array are cleared when all DomUs complete polling. Then, the next polling period starts.

In a round of scheduling, array *credit[n]* records the value of credits that VMCG assigns to each DomU. We check the value of *credit1[i]*, and then allocate credit for *credit[i]* according to Eq. (2) (Step 14). When a GPU command request block in the queue is dispatched, the corresponding value of credits is decremented by 1. Each vGPU has only 2 states: 1 or 0, where 1 indicates that the queue has credits while 0 denotes that the queue has run out of credits. Array *status[n]* is used to record the states (Step 15 to 23). First, if credit value is positive, i.e., the corresponding status equals 1, VMCG finds the corresponding DomU ID from array *DomID_Num[n]*, where n represents the number of running DomUs (Step 26). Then, for improving the performance of disk I/O, we reserve the strategy of CFQ in which four requests are dispatched each time [10]. So the first four GPU command request blocks in the pending queue are sent to vGPU scheduler according to the DomU ID (Step 27). Finally, the corresponding credit value is decreased by 1 (Step 28). Otherwise, the corresponding status value is 0, and the GPU command request blocks in the pending queue have to wait for the redistribution of credits (Step 30 and 31). VMCG reallocates credits for a queue that has no credits after polling before continuing the next round of scheduling.

6 Experimental Evaluation

In this section, we run a set of experiments to evaluate the performance of VMCG on an Xen-hosted platform by using multiple types of GPU workloads, which demonstrates that VMCG has several advantages in resource allocation compared with other approaches. The workloads include real graphic applications based on OpenGL and DirectX, and real parallel computing applications.

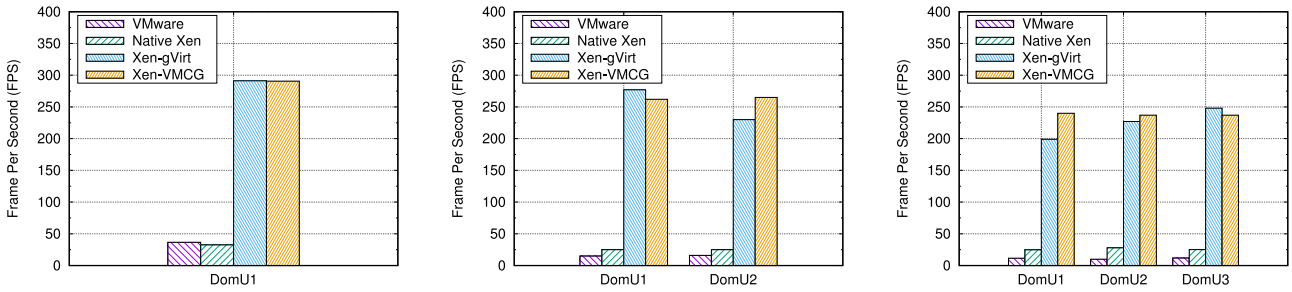
6.1 Experimental Setup

The experimental platform is an Intel i5-4590 4-core processor running at 3.2 GHz, with 16 GB of RAM, 1T hard disk, and an Intel HD4600 with graphics driver version win32_153618 (since gVirt only works with Intel graphics, our VMCG currently focuses on Intel graphics). We install Ubuntu 14.04 based on Linux kernel 3.18.0 in Host, and Windows 7 in each DomU. The Xen version is 4.5.0. Each DomU has an installed graphics driver. We allocate 1 vCPU and 2 GB of system memory for each DomU. A wide variety of application workloads or testing tools are utilized, such as 2D/3D real graphic applications based on OpenGL, parallel computing applications based on OpenCL, and DirectX application (e.g., 3DMark and Heaven3D).

Algorithm 1. Virtual Multi-channel Fair-scheduling

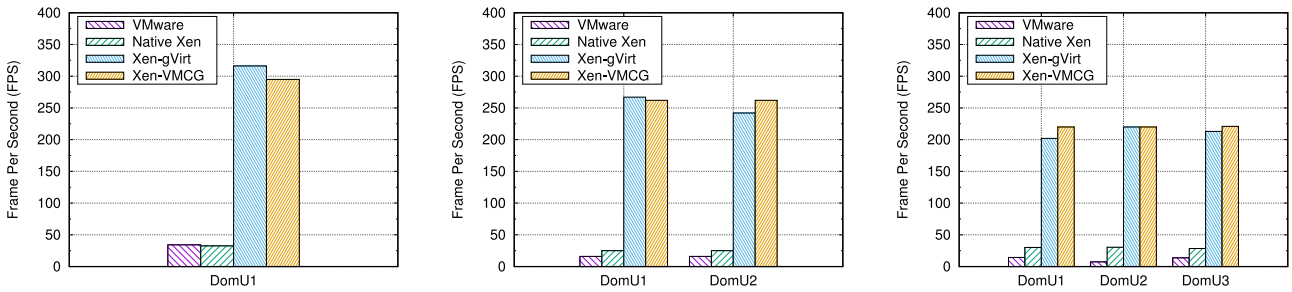
```
Data: Initialize the array:
      cpu_scheduler_time[n](cpu_sched_t[j], for short)
1  for i = 1 to n do
2      for j = i + 1 to n do
3          if cpu_sched_t[j] > cpu_sched_t[j-1]then
4              Exchange cpu_sched_t[j] and cpu_sched_t[j-1];
5              Exchange DomID_Num[j] and DomID_Num[j-1];
6          end
7      end
8  end
9  Allocate 16 ms rendering engine usage time for
   DomID_Num[1];
10 for k = 1 to n do
11     Allocate time slice for DomID_Num[k];
12     t_k = (cpu_sched_t[k])/(cpu_sched_t[1]) × 16ms;
13 end
14 Allocate credits after checking the value of credit1[i] accord-
   ing to Eq. (2);
15 for i = 1 to n do
16     if credit[i] > 0 then
17         /*1 indicates that GPU resources can be allocated for
           the queue */;
18         status[i] = 1;
19     else
20         /*0 indicates that GPU resources cannot be allocated
           for the queue */;
21         status[i] = 0;
22     end
23 end
24 for i = 1 to n do
25     if status[i] == 1 then
26         Find the DomU ID from DomID_Num[i];
27         Send out the first 4 GPU command request blocks to
           vGPU scheduler according to the DomU ID;
28         Decrease the corresponding credit values by 1;
29     else
30         Continue the next round;
31         Check the value of the next status;
32     end
33 end
34 Clear the array and continue to next round scheduling;
```

In Section 6.4, we have a comparison with the rCUDA middleware [26], [27]. Therefore, an NVIDIA GTX 750Ti has been installed in the above testbed which is as the server side of rCUDA. For the client side, three desktops with Intel Pentium G630 2-core processor running at 2.7 GHz, 4 GB of RAM, 1T hard disk are used. Ubuntu 14.04 is installed along with CUDA 8.0 (NVIDIA driver 390.67) and rCUDA



(a) 3D teapot, VM=1. (b) 3D teapot, VM=2. (c) 3D teapot, VM=3.

Fig. 5. Frame rate of 3D teapot.



(a) 3D gear, VM=1. (b) 3D gear, VM=2. (c) 3D gear, VM=3.

Fig. 6. Frame rate of 3D gear.

v16.11.04.02. The network between the server and client is an Ethernet with TCP/IP over 10 Gbps.

6.2 Graphics Rendering Performance

For evaluating graphic rendering performance, we run 3D teapot and 3D gear (an OpenGL demo application [19]) in VMware (including VMware VDI),[2] Native Xen, gVirt, and VMCG, respectively; the number of DomUs increases from 1 to 3 (the same priority for each DomU). Then, we replace the 3D teapot and 3D gear with DirectX workloads to redo the experiment.

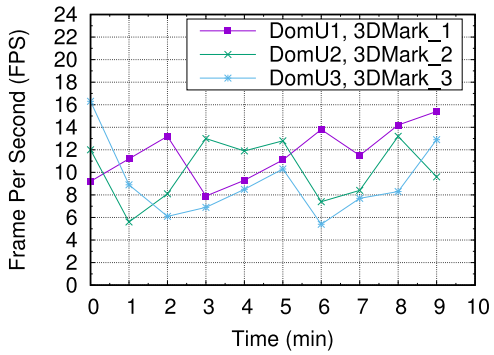
The experimental results are shown in Figs. 5 and 6. From the figures, we can observe that the performance of gVirt and VMCG is significantly higher than that of VMware and Native Xen. Comparing with VMware and Native Xen, the graphic rendering performance of VMCG is increased by approximately 2159 and 815 percent, respectively. This is because VMware and Native Xen do not enable GPU hardware acceleration, while VMCG and gVirt do. Compared with gVirt, VMCG almost achieves the same performance with gVirt for the 3D gear workload, but VMCG improves performance by 10 percent on average for the 3D teapot workload. The main reason for this is that VMCG has a better balance on different demands of GPU resources between DomUs, and 3D surface rendering and texture for the 3D teapot is much more GPU intensive than 3D gear. Moreover, we also run three processes of 3D teapot in the Native Domain; the total frames per second (FPS) is approximately 744 with GPU acceleration, which means that the performance (714 FPS) of VMCG is approximately 96 percent of that of the native GPU with gVirt's pass-through.

2. VMware also claims relatively high performance VDI implementations. So we compare the graphic rendering performance with VMware VDI in this section.

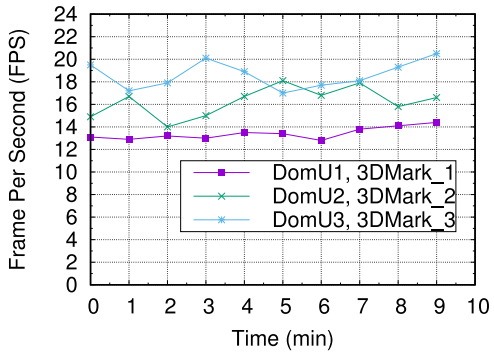
The experimental results for 3DMark (DirectX applications) [20] in three DomUs (the same priority for each DomU) are shown in Fig. 7. The frame rate of each DomU in gVirt and VMCG is not significantly different, but the performance curve of VMCG is smoother than that of gVirt. This is because VMCG relieves mutual interference between GPU command request blocks of DomUs by introducing a separated V-Channel and GPU command request queue for each DomU, and because the GPU fair-scheduling algorithm in the VMCG can efficiently balance the demands of GPU resources between multiple DomUs. We replace 3DMark with Heaven3D [21] and PassMark Simple [22] to redo the experiment. First, we run PassMark or Heaven3D workloads in one DomU, respectively. As shown in Fig. 8, the average performance for VMCG is 312 FPS (325 FPS in native windows 7) and 15 FPS (17 FPS in native windows 7). In the meanwhile, the performance varies between 309 FPS and 314 FPS for PassMark and between 14 FPS and 15 FPS for Heaven3D, but for gVirt, the performances vibrate in 258-315 FPS and 8-16 FPS. Then, we run three PassMark workloads in three DomUs (the same priority for each DomU), respectively. The experimental results are presented in Fig. 9. For gVirt, three DomUs obtain 81-122 FPS, 109-284 FPS, and 30-46 FPS, respectively. For VMCG, they obtain 110-115 FPS, 109-113 FPS, and 109-110 FPS, respectively. It can be observed that VMCG improves the fairness of GPU resource allocation. Nonetheless, VMCG enhances the fairness and stability of GPU resource allocation between multiple DomUs. We describe the situation in the following sections.

6.3 Fairness

Similar with the definition of fairness in [15] and [9], we use λ to denote the fairness metric of GPU resource allocation for multiple DomUs, which indicates the difference between the actually acquired throughputs (the GPU

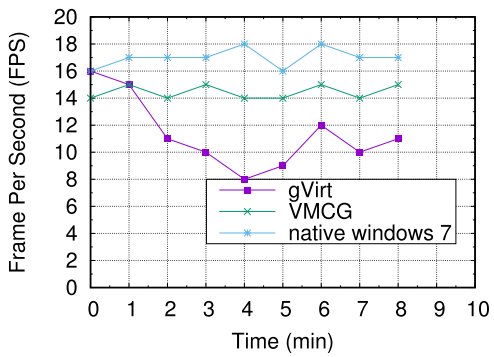


(a) gVirt

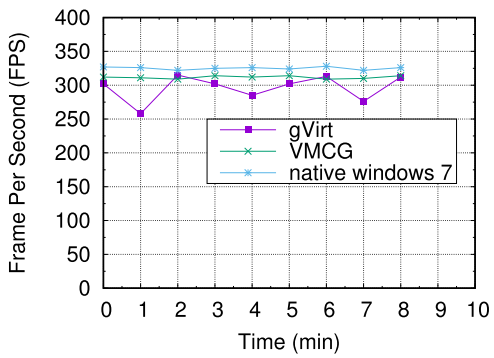


(b) VMCG

Fig. 7. Performance of 3DMark.



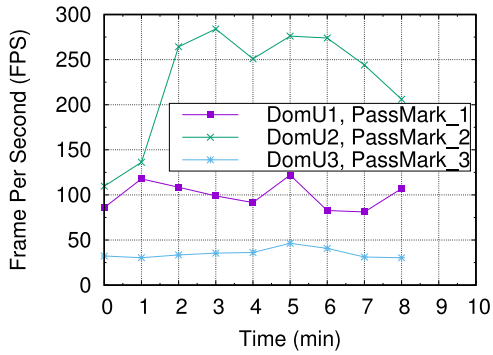
(a) Heaven 3D



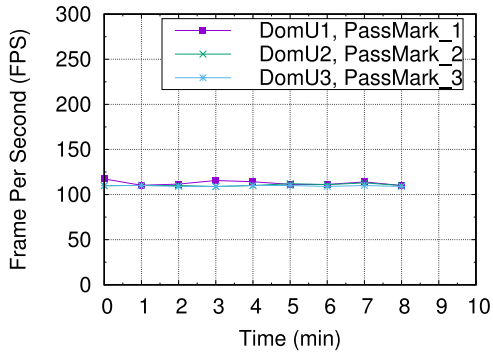
(b) PassMark Simple

Fig. 8. Performance of Heaven3D and PassMark in 1 DomU.

command request blocks capacity processed by the vGPU) and fairly allocated throughputs for each DomU. Let $DomU_i$ obtain a throughput G_i over an interval $[t_1, t_2]$. The



(a) gVirt



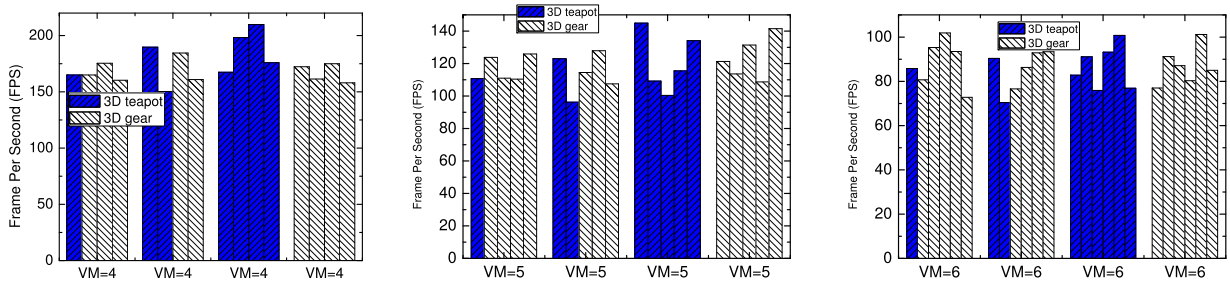
(b) VMCG

Fig. 9. Performance of PassMark in 3 DomUs.

total throughput is $G = \sum_{i=1}^{n} G_i$, and the measured weight of $DomU_i$ is $w_i' = G_i/G$. The fairness metric is defined as $\lambda = \sum_{i=1}^{n} |w_i - w_i'|$ during all time intervals $[t_1, t_2]$. A larger value of λ means worse fairness, since it means that the ratio of the DomU's throughputs have a bad match with the weights.

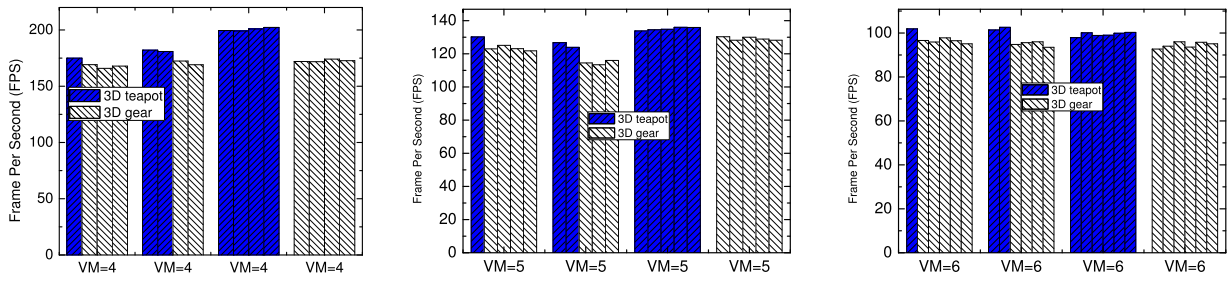
Taking Fig. 5c running OpenGL workloads in gVirt and VMCG as an example, let the desired weights be in the ratio 1:1:1, then, the vector of specified weights (expected from a fair schedule) is $W = [w_1, w_2, w_3] = [1/3, 1/3, 1/3]$. The measured throughputs for the three DomUs using gVirt and VMCG are (199.0, 227.0, 248.0) and (240.6, 237.1, 237.0), respectively. The vector of measured weights is $W' = [w_1', w_2', w_3'] = [199/674, 227/674, 248/674] = [0.29, 0.34, 0.37]$ for gVirt, and $W' = [w_1', w_2', w_3'] = [240.6/714.7, 237.1/714.4, 237/714.1] = [0.34, 0.33, 0.33]$ for VMCG. Hence the fairness metric is $\lambda = 0.09$ for gVirt, and $\lambda = 0.01$ for VMCG. Regarding the running 3DMark workloads in gVirt and VMCG (Fig. 7), the fairness metric is 0.41 and 0.15, respectively. Both OpenGL and DirectX workloads indicate that the fairness of VMCG is better than that of gVirt.

In order to further evaluate the effectiveness of VMCG for fairness, we extend the number of DomUs to 4, 5, and 6. Experimental results for gVirt and VMCG are shown in Figs. 10 and 11. When the same workloads are deployed to all DomUs, e.g., 3D teapot or 3D gear (another OpenGL demo application), the obtained frame rate for each DomU in gVirt differs greatly; however, it remains almost equal across DomUs in VMCG. Then, we run hybrid OpenGL workloads, for example, a 3D teapot in 1 or 2 of DomUs and other DomUs running 3D gear. From this experiment, we observe that those DomUs running



(a) gVirt, VM=4. (b) gVirt, VM=5. (c) gVirt, VM=6.

Fig. 10. Performance of gVirt.



(a) VMCG, VM=4. (b) VMCG, VM=5. (c) VMCG, VM=6.

Fig. 11. Performance of VMCG.

the same workloads have an obvious difference for gVirt, but obtain almost the same frame rate under VMCG. The main reason for this is that the VMCG dispatches GPU command request blocks from the corresponding DomUs according to a fair-scheduling algorithm. Moreover, the rendering performance of a 3D teapot is better than that of 3D gear because longer time slices are assigned for the DomUs running the 3D teapot.

To better understand this fairness, we here elaborate the internals of the VMCG framework, such as transferring credits and GPU command request block percentages per DomU. Taking three DomUs which own the same priorities (the desired weight ratio is 1:1:1) as an example, when one DomU runs a 3D teapot and the other two DomUs run 3D gear, the percentage of GPU command request block in both DomU2 and DomU3 is 33 percent, and is 34 percent in DomU1. According to Eq. (2), VMCG allocates 34, 33, and 33 credits to DomU1, DomU2, and DomU3 respectively, i.e., *credit1* [1] = 34, *credit1* [2] = 33, and *credit1* [3] = 33. After GPU command request blocks of *DomUi* are dispatched out from the corresponding pending queue, its credits is decreased. The number of decreasing credits is determined by the number of dispatching GPU command request blocks. Lastly, the VMCG allocates 16 ms, 15 ms, and 15 ms to DomU1, DomU2, and DomU3, respectively, for GPU rendering.

The fairness metric is shown in Fig. 12. It can be seen that the fairness metric of VMCG is significantly lower than that of gVirt. Furthermore, the VMCG improves fairness by approximately 72.5, 77.8, 79.1, 80.7, and 82.6 percent compared to gVirt under 2, 3, 4, 5, and 6 DomUs, respectively. This is because the VMCG mitigates mutual interference between DomUs during competition for GPU resources through exploiting multiple V-Channels and the fair-scheduling algorithm. Moreover, with an increasing number of DomUs, the fairness metric of VMCG tends to be smooth and steady, while it becomes sharper in gVirt.

Finally we compare the fairness between NVIDIA GRID [23] and VMCG. GRID is a virtualization solution which is based on NVIDIA vGPU to share the power of NVIDIA GPUs across VMs. We run Heaven3D workload in three DomUs for GRID and VMCG respectively and a Tesla M60 GPU graphics card is used to support the GRID's experiment. The evaluation results are presented in Fig. 13. From the graph, it is apparent that the performance of GRID is better than that of VMCG, this is due to the powerful graphics processing capability of the Tesla GPUs. But our focus here is the fairness between two methods. Obviously, the performance curve of VMCG in Fig. 13b is smoother than that of gVirt in Fig. 13a. Then we calculate the fairness metric of GRID and VMCG according to Fig. 13. The average fairness index is $\lambda = 0.088$ for GRID and $\lambda = 0.057$ for VMCG respectively which is improved by nearly 35.2 percent, it can be seen that the fairness metric of VMCG is better than that of GRID.

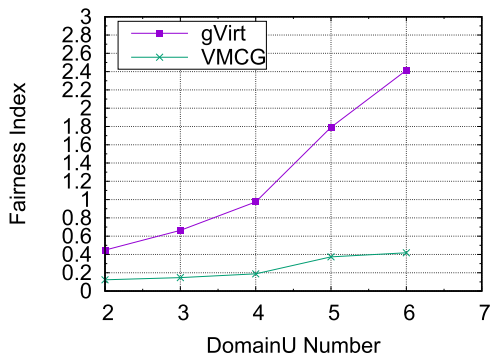
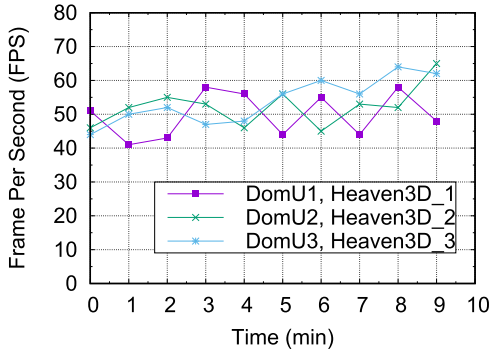
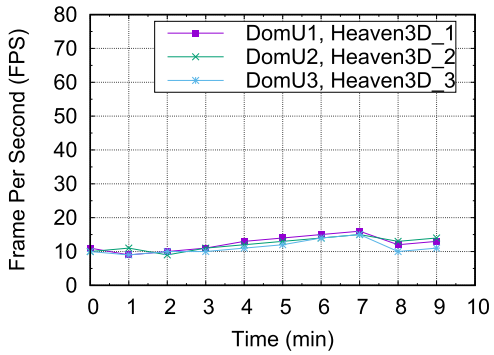


Fig. 12. Fairness for VMCG and gVirt.



(a) GRID



(b) VMCG

Fig. 13. Performance of Heaven3D in 3 DomUs.

6.4 Real Parallel Computing for VMCG

For evaluating real parallel computing applications, the first application case we consider is real-time high definition (HD, the resolution is 1920x1080) video dehazing. By exploiting the parallel computing capacity of GPU, we designed a parallel optimization method [33] (denoted by OpenCL_dehazing) for the real-time dehazing of HD hazing videos based on a single image haze removal algorithm [34], [35]. We deploy one DomU, and test OpenCL_dehazing under Original Xen, gVirt, and the VMCG. Original Xen only obtains about 5 FPS because it exploits vCPU for computing instead of GPU; meanwhile, the gVirt and VMCG achieve 22 FPS and 25 FPS, respectively, indicating that the parallel computing performance with GPU acceleration in the virtualization computing environment can be improved by about 500 percent. To dehaze three hazing videos at the same time, three OpenCL_dehazing workloads are deployed to three DomUs; the results are shown in Fig. 14. Although the same dehazing effect for gVirt and the VMCG is acquired, the VMCG obtains more fair and stable performance (the fairness metric for gVirt and VMCG is 0.13 and 0.025, respectively, corresponding to an improvement of 81 percent). This indicates that the VMCG is adaptable to parallel computing workloads.

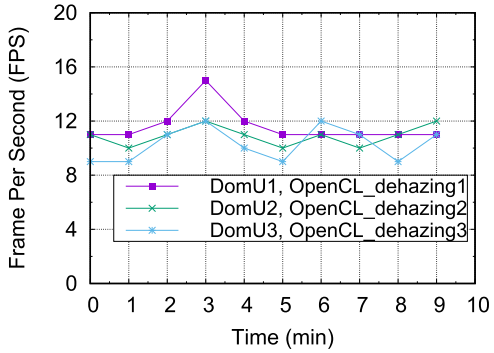
The second application case utilizes Rodinia [36], [37], which is used widely by academia. From all the 21 benchmarks in the suite, we selected 14 typical parallel computing benchmarks of different work-group sizes which are broadly applied to all kinds of applications. The performance for these parallel computing workloads is measured by the time consumption of the same data processing concurrently in each DomU. We set three DomUs to evaluate the benchmarks under gVirt and the VMCG. The results are shown in Table 2. Here, the time consumption of three DomUs is almost uniform under the VMCG and is also less than that of gVirt in average. This is because the VMCG ensures fair GPU resource allocation among DomUs so that it reduces the cost of useless time (such as blocking time). Meanwhile, according to the evaluation method in Section 5.2, the average fairness of the VMCG in Table 2 increases by approximately 76 percent than that of gVirt. Then we compare the fairness between rCUDA and VMCG. The experimental results of rCUDA are also presented in Table 2. Because of the different platforms, we just focus on the fairness metrics which are improved by 71.6 percent on average compared to rCUDA, respectively. The reason is that the VMCG fairly allocates GPU resources for DomUs running parallel computing workloads.

6.5 Stability of VMCG

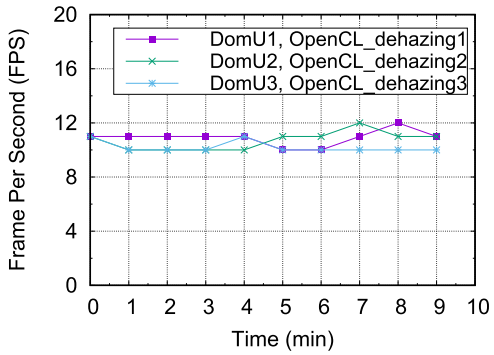
The stability of GPU resource allocation is also tested in this section. We obtain the experimental results by running the

TABLE 2
Experimental Results of Rodinia for gVirt, VMCG and rCUDA

Bench.	gVirt				VMCG				rCUDA			
	Time(s)			Fairness (λ)	Time(s)			Fairness(λ)	Time(s)			Fairness(λ)
	DomU1	DomU2	DomU3		DomU1	DomU2	DomU3		DomU1	DomU2	DomU3	
Srad	1.78	1.506	1.466	0.078	1.448	1.412	1.4	0.013	0.325	0.462	0.309	0.156
Backprop	0.256	0.19	0.253	0.135	0.206	0.216	0.212	0.017	0.319	0.371	0.294	0.083
Hotspot3D	0.436	0.683	0.347	0.232	0.373	0.345	0.365	0.03	0.354	0.346	0.239	0.179
Nw	0.403	0.316	0.393	0.106	0.377	0.393	0.384	0.014	0.568	0.76	0.777	0.14
Bfs	1.978	2.052	1.726	0.07	1.658	1.694	1.596	0.02	2.736	2.828	2.247	0.098
LavaMD	3.063	3.065	2.066	0.185	2.239	2.124	2.102	0.026	0.853	0.636	0.981	0.169
Leukocyte	4.858	5.292	4.876	0.037	4.803	4.956	4.911	0.012	1.125	1.07	0.981	0.051
Particlefilter	0.202	0.24	0.195	0.083	0.202	0.204	0.189	0.032	0.187	0.218	0.219	0.069
Nn	0.132	0.171	0.153	0.093	0.162	0.139	0.158	0.063	0.184	0.208	0.158	0.097
B+tree	0.207	0.168	0.165	0.093	0.164	0.171	0.168	0.015	0.285	0.224	0.334	0.147
Dwt2d	0.451	0.418	0.369	0.074	0.458	0.434	0.44	0.021	0.251	0.271	0.23	0.057
Hotspot	1.172	1.127	1.172	0.018	1.144	1.13	1.123	0.007	0.442	0.437	0.421	0.019
Pathfinder	0.302	0.284	0.338	0.062	0.296	0.287	0.295	0.013	0.376	0.39	0.399	0.021
Gaussian	0.287	0.199	0.197	0.154	0.194	0.193	0.199	0.012	0.15	0.108	0.193	0.211



(a) gVirt, OpenCL.



(b) VMCG, OpenCL.

Fig. 14. Parallel computing for video dehazing.

same 2D/3D application workload in three DomUs. Fig. 15 shows the relationship between them.

As seen from the figure, the stability of VMCG is better than Native Xen and gVirt. Taking 3D teapot as an example, DomU1, DomU2, DomU3 all achieve almost the same frame rate (approximately 238 FPS) for VMCG. For gVirt, the frame rate fluctuates over time and ranges from 190 to 250 FPS. Compared with that of Native Xen and gVirt, VMware also shows better stability, close to that of our VMCG, but its overall performance is lower than that of VMCG. In general, when DomUs run on the same physical host, VMCG can significantly improve the stability of GPU resource allocation and the stability of VMCG remains invariant when increasing the number of DomUs. The reason is that VMCG relieves mutual interference between GPU command request blocks of DomUs through the V-Channel. We can conclude that VMCG can lower the interference among multiple DomUs by introducing a separated V-Channel and GPU command request queue for each DomU.

6.6 Support for Hybrid Workloads

In this section, we evaluate VMCG's support for hybrid workloads using OpenGL 3D simulated flight application [19] (OpenGL_flight), parallel computing application based on OpenCL [33], [34], [35] (e.g., foggy video dehazing using OpenCL, denoted by OpenCL_dehazing, the resolution is 1024x768), and DirectX application [20], [21] (e.g., 3DMark and Heaven3D).

We build two sets of hybrid workloads in three DomUs, and configure two sets of priorities for these DomUs, i.e., the same priorities (1:1:1) and different priorities (4:2:1).

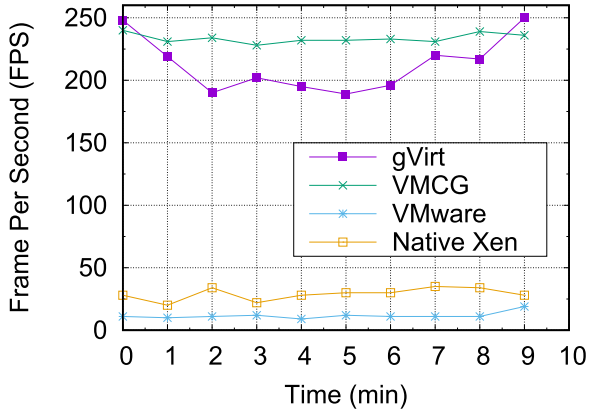
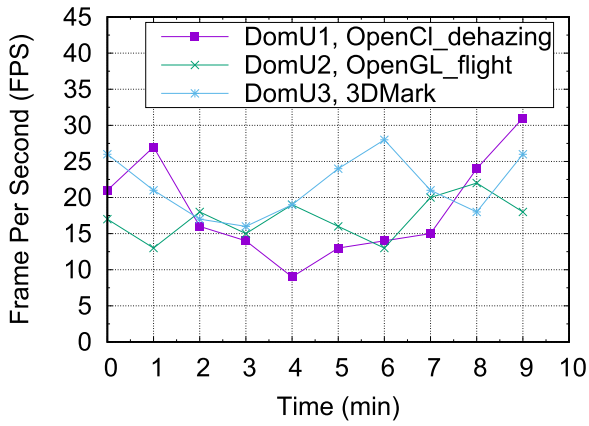
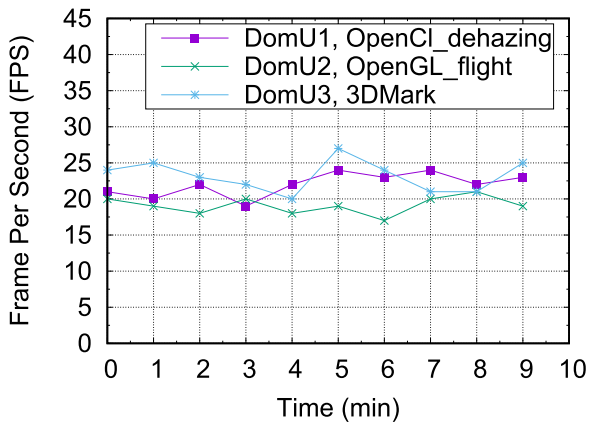


Fig. 15. Stability of GPU resource allocation between multiple DomUs.

Figs. 16 and 17 present the experimental results. Compared with Figs. 16b and 17b, the performance of three DomUs has a very obvious fluctuation in gVirt, but VMCG obtains more stable performance distribution. The fairness of the VMCG increases by approximately 75 and 78 percent than that of gVirt for two sets of hybrid workloads and priorities. The main reason for this is that the VMCG eliminates unfair competition between DomUs, which ensures stable GPU resource allocation. Thus, we conclude that the VMCG also improves the fairness of GPU resource allocation in

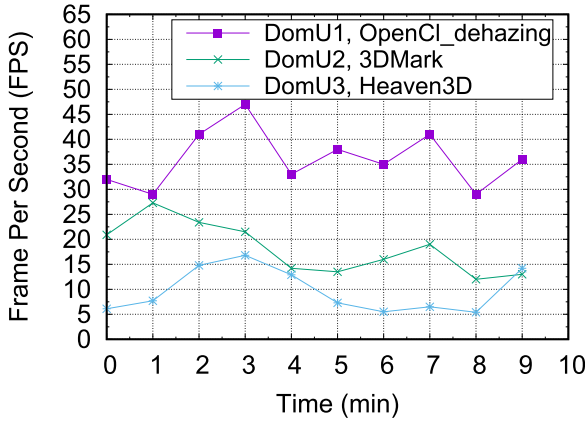


(a) gVirt

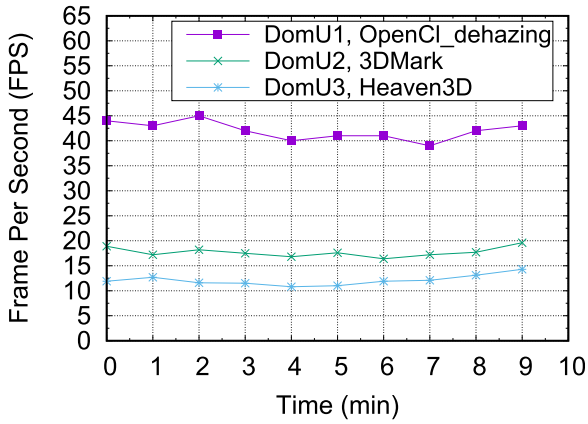


(b) VMCG

Fig. 16. Hybrid workloads in 3 DomUs (with the same priorities).

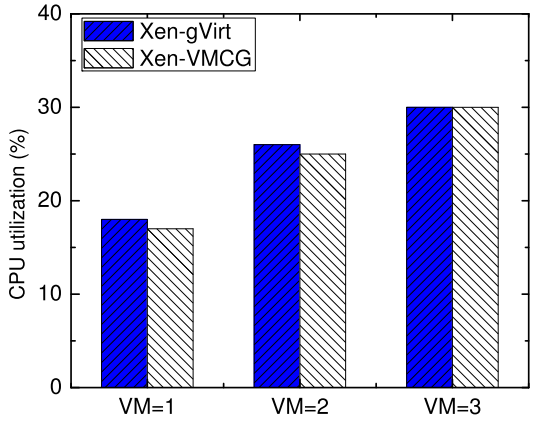


(a) gVirt

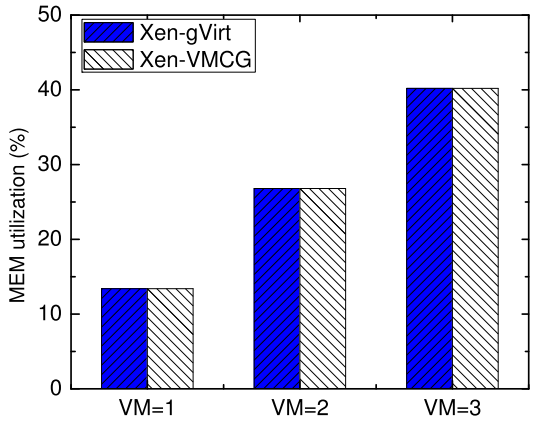


(b) VMCG

Fig. 17. Hybrid workloads in 3 DomUs (with different priorities).



(a) CPU utilization between gVirt and VMCG.



(b) MEM utilization between gVirt and VMCG.

Fig. 18. Overhead for running 1-3 DomUs.

virtualization computing environments with hybrid application workloads.

6.7 Overhead

In this part, we compare the overhead of the same application workload with VM=1, VM=2, and VM=3, respectively in gVirt and VMCG. Fig. 18 shows the utilization of host CPU and memory. From the figure, when increasing the number of VMs, the utilization of CPU and MEM becomes greater, e.g., MEM utilization of VM=3 is twice than that of VM=1 for both gVirt and VMCG. The reason is that the GPU command request blocks from creating new VMs must consume additional CPU and MEM for dispatching. However, memory utilization is the same between gVirt and VMCG when VM=1, 2, and 3, and CPU utilization of VMCG is also mostly equal to that of gVirt. In other words, VMCG does not add extra memory and CPU consumption; i.e., the design of VMCG does not induce more overhead.

7 Related Work

Our research aims at fair GPU resource allocation on the premise of adapting common GPU application workloads in VMs. This section presents the related work.

GPU Application Workload Support in VM. A GPU's application workload mainly includes graphic rendering and parallel computing. To support graphic rendering in VM, many studies [3], [5], [6], [7], and [8] proposed some graphics processing methods for graphic application workloads in multiple DomUs. However, the parallel computing potential and hardware accelerated power have not be considered in virtualization environments. It is also very important for improving performance to have a fair GPU allocation between DomUs. A GPU's parallel computing ability in VMs was researched [4], [11], [14], and it is common to operate multiple types of GPU workloads simultaneously in VMs, for example, in multi-screen independent 2D/3D graphic rendering and hybrid GPU-intensive applications. Our VMCG supports hybrid application workloads (2D/3D graphic rendering, real parallel computing, and hybrid workloads) that are distributed to multiple DomUs hosted in the same physical GPU.

GPU Resource Allocation. The BAND scheduling algorithm [17], [18] was designed by extending CREDIT scheduling in GPUvm [4]; however, a DomU with a low priority never obtains GPU, and inserting waiting time does not achieve real fairness when each DomU needs different GPU resources according to its respective requirement. In our VMCG, GPU resources are dynamically allocated and adjusted by the credit allocating and transferring mechanisms. A credit allocation approach was provided by

Suzuki et al. [4] and Kato et al. [17], but they ignored the number of credits allocated in each round of scheduling. Therefore, we remodel credit, including credit transferring and allocating. In addition, the overhead of memory and CPU in the work by [4] was much higher because of the use of physic and virtual indexes.

GPU Scheduling. GPU scheduling based on fairness can generally be divided into two categories. The first type is the scheduling based on API. An API structure framework is porposed which all types of scheduling algorithms can be integrated into [6] and [7]. Thus, which scheduling algorithm is more suitable for DomUs in a time period might be flexibly determined. This hybrid scheduling has global monitoring, which obtains information whenever DomUs must leverage GPU. The algorithm belongs to fair-scheduling from a macro point of view. However, it requires the support of many specific algorithms, such as some scheduling algorithms mentioned above, which makes the system particularly complex.

Then the second category is the scheduling based on vGPU. The application of 3D rendering is achieved through a GPU rendering engine, which becomes a very complex part of a GPU. Blindly virtualizing a rendering engine is almost unrealistic. A vGPU structure is designed by which each DomU could access the physical GPU [2], [4], [6]. The vGPU was a software simulation of GPU, and the vGPU scheduling algorithm served as a bridge connecting the DomU and physical GPU. In our VMCG, we redesign a fair-scheduling algorithm based on the actual demands of GPU resources after remodeling credit allocating and transferring. This redesigned algorithm is aimed at fairly allocating GPU resources across multiple DomUs to approximate native GPU performance.

8 Conclusion

In this paper, we have presented a design and implementation of a VMCG for fairly allocating GPU resources in a virtualization computing environment. Our VMCG can effectively mitigate mutual interference between DomUs, fairly allocate GPU resources, and improve performance. When multiple DomUs concurrently contend for GPU resources, the GPU command request blocks from multiple DomUs are aggregated into a unified queue in Dom0, which cannot ensure independence and fairness. However, the VMCG provides a separate V-Channel for each DomU in Dom0; thus, the GPU command request blocks are sent into corresponding V-Channels according to their own DomU ID. To fairly allocate GPU resources for each DomU, we remodel the credit allocating and transferring mechanisms, and redesign the GPU fair-scheduling algorithm. Finally, we evaluate the fairness of GPU resource allocation for the VMCG with a series of experiments. Experimental results show the effectiveness of VMCG. Furthermore, the VMCG is dependent upon software implementation and has strong portability.

Acknowledgments

The authors are grateful to the anonymous reviewers for their helpful feedback. This research was supported by the National Natural Science Foundation of China under grants 61672218 (Virtual Multi-channel Asymmetry Parallel Model and Fair Scheduling Scheme for GPU).

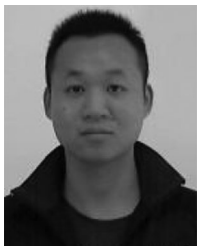
References

[1] M. D. Carroll, I. Hadzic, and W. A. Katsak, "3D rendering in the cloud," *Bell Labs Tech. J.*, vol. 17, no. 2, pp. 55–66, 2012, doi: 10.1002/bltj.21544.
[2] K. Tian, Y. Dong, and D. Cowperthwaite, "A full GPU virtualization solution with mediated pass-through," in *Proc. USENIX Annu. Tech. Conf.*, Jun. 19–20, 2014, pp. 121–132.
[3] Y. Joo, D. Lee, and Y. Ik Eom, "Delegating OpenGL commands to host for hardware support in virtualized environments," in *Proc. 8th Int. Conf. Ubiquitous Inf. Manage. Commun.*, Jan. 9–11, 2014, Art. no. 95.
[4] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "GPUvm: Why not virtualizing GPUs at the hypervisor?" in *Proc. USENIX Annu. Tech. Conf.*, 2014, pp: 109–122.
[5] C. Smowton, "Secure 3D graphics for virtual machines," in *Proc. 2nd Eur. Workshop Syst. Security*, 2009, pp. 36–43.
[6] Z. Qi, J. Yao, C. Zhang, M. Yu, Z. Yang, and H. Guan, "VGRIS: Virtualized GPU resource isolation and scheduling in the gaming," *ACM Trans. Archit. Code Optimization*, vol. 11, no. 2, Jun. 2014, Art. no. 17.
[7] M. Dowty and J. Sugreman, "GPU virtualization on VMware's hosted I/O architecture," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 3, pp. 73–82, 2009.
[8] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "GViM: GPU-accelerated virtual machines," in *Proc. 3rd Workshop Syst.-level Virtualization High Perform. Comput.*, Mar. 31, 2009, pp. 17–24.
[9] H. Tan, L. Huang, Z. He, Y. Lu, and X. He, "DMVL: An I/O bandwidth dynamic allocation method for virtual networks," *J. Netw. Comput. Appl.*, vol. 39, pp. 104–116, 2013.
[10] H. Tan, C. Li, Z. He, K. Li, and K. Hwang, "VMCD: A virtual multi-channel disk I/O scheduling method for virtual machines," *IEEE Trans. Serv. Comput.*, vol. 9, no. 6, pp. 982–995, Nov./Dec. 2016, doi: 10.1109/TSC.2015.2436388.
[11] L. Vu, H. Sivaraman, and R. Bidarkar, "GPU virtualization for high performance general purpose computing on the ESX hypervisor," in *Proc. High Perform. Comput. Symp.*, Apr. 13–16, 2014, Art. no. 2.
[12] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. War eld., "Xen and the art of virtualization," in *ACM SIGOPS Operating Systems Review*, vol. 37, pp. 164–177, 2003.
[13] M. Pharr and R. Fernando, *GPU Gems Programming Techniques, Tips, and Tricks for Real-Time Graphics*, 1st Ed. Boston, MA, USA: Addison-Wesley, 2007, pp. 320–400.
[14] L. Shi, H. Chen, and J. Sun, "vCUDA: GPU accelerated high performance computing in virtual machines," *IEEE Trans. Comput.*, vol. 61, no. 6, pp. 804–816, Jun. 2012.
[15] A. Gulati, A. Merchant, M. Uysal, P. Padala, and P. Varman, "Efficient and adaptive proportional share I/O scheduling," *ACM Sigmetrics Perform. Eval. Rev.*, vol. 37, no. 2, pp. 79–80, 2009.
[16] C. Rossbach, et al., "PTask: Operating system abstractions to manage GPUs as compute devices," in *Proc. 23th ACM Symp. Operating Syst. Principles*, 2011, pp. 233–248.
[17] S. Kato, M. Mcthrow, C. Maltzahn, and S. Brandt, "Gdev: First-class GPU resource management in the operating system," in *Proc. USENIX Conf. Annu. Tech. Conf.*, Jun. 13–15, 2012, pp. 401–412.
[18] K. Menychtas, K. Shen, and M. L. Scott, "Disengaged scheduling for fair, protected access to fast computational accelerators," in *Proc. 19th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2014, pp. 301–316.
[19] mesa demos. 2006. [Online]. Available: https://cgit.freedesktop.org/mesa/demos
[20] 3DMark. 2017. [Online]. Available: http://www.futuremark.com
[21] Heaven3D. 2017. [Online]. Available: http://unigine.com/products/heaven
[22] PassMark2D. 2016. [Online]. Available: http://www.passmark.com
[23] GRID. 2017. [Online]. Available: http://www.nvidia.com/grid
[24] Rodinia. 2017. [Online]. Available: http://lava.cs.virginia.edu/Rodinia/download_links.htm

[25] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *Proc. Int. Conf. High Perform. Comput. Simul.*, 2010, pp. 224–231.
[26] C. Reaño and F. Silla, "A performance comparison of CUDA remote GPU virtualization frameworks," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2015, pp. 488–489.
[27] C. Reaño, F. Silla, G. Shainer, and S. Schultz, "Local and remote GPUs perform similar with EDR 100G InfiniBand," in *Proc. Ind. Track 16th Int. Middleware Conf.*, 2015, Art. no. 4.
[28] F. Pérez, C. Reaño, and F. Silla, "Providing CUDA acceleration to KVM virtual machines in infiniband clusters with rCUDA," in *Proc. Distrib. Appl. Interoperable Syst.*, 2016, pp. 82–95.
[29] J. Prades, C. Reaño, and F. Silla, "CUDA acceleration for xen virtual machines in infiniband clusters with rCUDA," *ACM SIGPLAN Notices*, vol. 51, no. 8, 2016, Art. no. 35.
[30] J. Duato, A. J. Pena, F. Silla, J. C. Fernandez, R. Mayo, and E. S. Quintana-Orti, "Enabling CUDA acceleration within virtual machines using rCUDA," in *Proc. 18th Int. Conf. High Perform. Comput.*, 2011, pp. 1–10.
[31] J. Prades and F. Silla, "Turning GPUs into floating devices over the cluster: The beauty of GPU migration," in *Proc. 46th Int. Conf. Parallel Process. Workshops*, 2017, pp. 129–136.
[32] A. J. Peña, C. Reaño, F. Silla, R. Mayo, E. S. Quintana-Ortí, and J. Duato, "A complete and efficient CUDA-sharing solution for HPC clusters," *Parallel Comput.*, vol. 40, no. 10, pp. 574–588, 2014.
[33] H. Tan, X. He, Z. Wang, G. Liu, "Parallel implementation and optimization of high definition video real-time dehazing," *Multimedia Tools Appl.*, vol. 76, no. 22, pp. 23413–23434, 2017, doi: 10.1007/s11042-016-4036-4.
[34] K. He, J. Sun, and X. Tang, "Single image haze removal using dark channel prior," in *Proc IEEE Conf. Comput. Vis. Pattern Recognit.*, 2011, pp. 2341–2353.
[35] J. Tarel and N. Hautiere, "Fast visibility restoration from a single color or gray level image," in *Proc 12th IEEE Int. Conf. Comput. Vis.*, 2009, pp. 2201–2208.
[36] S. Che, M. Boyer, J. Meng, and D. Tarjan, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc IEEE Int. Symp. Workload Characterization*, 2009, pp. 44–51.
[37] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary CMP workloads," in *Proc. IEEE Int. Symp. Workload Characterization*, 2010, pp. 1–11.

Huailiang Tan received the BS degree from Central South University, China, in 1992, the MS degree from Hunan University, China, in 1995, and the PhD degree from Central South University, China, in 2001. He has more than eight years of industrial R&D experience in the field of information technology. He was a visiting scholar at Virginia Commonwealth University from 2010 to 2011. He is currently an associate professor with the College of Computer Science and Electronic Engineering, Hunan University, China. His research interests include high performance I/O, GPU architectures, and embedded systems.

Yanjie Tan received the BS and MS degrees from the Huazhong University of Science and Technology, China, in 2011, 2015, respectively. He is working toward the Dr degree student in the Department of Computer Science and Technology, Hunan University. His current research interests include flash-based storage systems, GPU virtualization and GPU architectures.



Xiaofei He received the BS degree in Computer Science and Technology from the Huaihai Institute of Technology, in 2014. He is working toward the MS degree student in the Department of Computer Science and Technology, Hunan University. His current research interests include GPU virtualization and embedded systems.

Kenli Li (SM'15) received the PhD degree in computer science from the Huazhong University of Science and Technology, Wuhan, China, in 2003. He was a visiting scholar with the University of Illinois at Urbana–Champaign, Champaign, IL, from 2004 to 2005. He is currently a full professor of Computer Science and Technology with Hunan University, Changsha, China, and the deputy director of the National Supercomputing Center, Changsha. He has published more than 130 research papers in international conferences and journals, such as the *IEEE Transactions on Computers*, the *IEEE Transactions on Parallel and Distributed Systems*, the *Journal of Parallel and Distributed Computing*, ICPP, and CCGrid. His current research interests include parallel computing, high-performance computing, and grid and cloud computing. He serves on the editorial board of the *IEEE Transactions on Computers*. He is an outstanding member of CCF.

Keqin Li is a SUNY distinguished professor of computer science. His current research interests include parallel computing and high-performance computing, distributed computing, energy-efficient computing and communication, heterogeneous computing systems, cloud computing, big data computing, CPU-GPU hybrid and cooperative computing, multicore computing, storage and file systems, wireless communication networks, sensor networks, peer-to-peer file sharing systems, mobile computing, service computing, Internet of things and cyber-physical systems. He has published more than 530 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He is currently or has served on the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Computers*, the *IEEE Transactions on Cloud Computing*, the *IEEE Transactions on Services Computing*, the *IEEE Transactions on Sustainable Computing*. He is an IEEE fellow.